

单元测试系统之-EasyMock

单元测试是 XP极力推荐的测试驱动开发模式，是保证软件质量的重要方法。尽管如此，对许多类的单元测试仍然是极其困难的，例如，对数据库操作的类进行测试，如果不准备好数据库环境以及相关测试数据，是很难进行单元测试的；再例如，对需要运行在容器内的 **Servlet**或**EJB**组件，脱离了容器也难于测试。幸运的是，**Mock Object**可以用来模拟一些我们需要的类，这些对象被称之为模仿对象，在单元测试中它们特别有价值。**Mock Object**用于模仿真实对象的方法调用，从而使得测试不需要真正的依赖对象。**Mock Object**只为某个特定的测试用例的场景提供刚好满足需要的最少功能。它们还可以模拟错误的条件，例如抛出指定的异常等。目前，有许多可用的 **Mock**类库可供我们选择。一些 **Mock**库提供了常见的模仿对象，例如：**HttpServletRequest**，而另一些 **Mock**库则提供了动态生成模仿对象的功能，本文将讨论使用 **EasyMock**动态生成模仿对象以便应用于单元测试。到目前为止，**EasyMock**提供了 1.2版本和 2.0版本，2.0版本仅支持 Java SE 5.0，本例中，我们选择 **EasyMock 1.2 for Java 1.3**版本进行测试，可以从 <http://www.easymock.org>下载合适的版本。我们首先来看一个用户验证的 **LoginServlet**类：

```
/**
 * LoginServlet.java
 * Author: Liao Xue Feng, www.javaedev.com
 */
package com.javaedev.test.mock;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LoginServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        String username = request.getParameter("username");
        String password = request.getParameter("password");

        // check username & password:
        if("admin".equals(username) && "123456".equals(password)) {
            ServletContext context = getServletContext();
            RequestDispatcher dispatcher = context.getNamedDispatcher("dispatcher");
            dispatcher.forward(request, response);
        }
        else {
            throw new RuntimeException("Login failed.");
        }
    }
}
```

```
}  
}  
}
```

这个 `Servlet` 实现简单的用户验证的功能，若用户名和口令匹配“admin”和“123456”，则请求被转发到指定的 `dispatcher` 上，否则，直接抛出 `RuntimeException`。为了测试 `doPost()` 方法，我们需要模拟 `HttpServletRequest`，`ServletContext` 和 `RequestDispatcher` 对象，以便脱离 `J2EE` 容器来测试这 `Servlet`。我们建立 `TestCase`，名为 `LoginServletTest`：

```
public class LoginServletTest extends TestCase {  
}
```

我们首先测试当用户名和口令验证失败的情形，演示如何使用 `EasyMock` 来模拟 `HttpServletRequest` 对象：

```
public void testLoginFailed() throws Exception {  
    MockControl mc = MockControl.createControl(HttpServletRequest.class);  
    HttpServletRequest request = (HttpServletRequest)mc.getMock();  
    // set Mock Object behavior:  
    request.getParameter("username");  
    mc.setReturnValue("admin", 1);  
    request.getParameter("password");  
    mc.setReturnValue("1234", 1);  
    // ok, all behaviors are set!  
    mc.replay();  
    // now start test:  
    LoginServlet servlet = new LoginServlet();  
    try {  
        servlet.doPost(request, null);  
        fail("Not caught exception!");  
    }  
    catch(RuntimeException re) {
```

```
assertEquals("Login failed.", re.getMessage());  
  
}  
  
// verify:  
  
mc.verify();  
  
}
```

仔细观察测试代码，使用 EasyMock来创建一个 Mock对象需要首先创建一个 MockControl：

```
MockControl mc = MockControl.createControl(HttpServletRequest.class);
```

然后，即可获得 MockControl创建的Mock对象：

```
HttpServletRequest request = (HttpServletRequest)mc.getMock();
```

下一步，我们需要“录制”Mock对象的预期行为。在 LoginServlet中，先后调用了 request.getParameter("username")和 request.getParameter("password")两个方法，因此，需要在 MockControl中设置这两次调用后的指定返回值。我们期望返回的值为“admin”和“1234”：

```
request.getParameter("username"); // 期望下面的测试将调用此方法，参数为  
"username"  
mc.setReturnValue("admin", 1); //期望返回值为  
"admin"，仅调用  
1次  
request.getParameter("password"); // 期望下面的测试将调用此方法，参数为  
" password"  
mc.setReturnValue("1234", 1); // 期望返回值为  
"1234"，仅调用
```

```
1次  
紧接着，调用  
mc.replay()，表示  
Mock对象“录制”完毕，可以开始按照我们设定的方式运行，  
我们对  
LoginServlet进行测试，并预期会产生一个  
RuntimeException：
```

```
LoginServlet servlet = new LoginServlet();  
  
try {  
  
servlet.doPost(request, null);
```

```
fail("Not caught exception!");  
}  
  
catch(RuntimeException re) {  
  
assertEquals("Login failed.", re.getMessage());  
}  
}
```

由于本次测试的目的是检查当用户名和口令验证失败后，LoginServlet是否会抛出

RuntimeException，因此，response对象对测试没有影响，我们不需要模拟它，仅仅传入null即可。

最后，调用

mc.verify()检查

Mock对象是否按照预期的方法调用正常运行了。

运行

JUnit，测试通过！表示我们的

Mock对象正确工作了！

下一步，我们来测试当用户名和口令匹配时，

LoginServlet应当把请求转发给指定的

RequestDispatcher。在这个测试用例中，我们除了需要HttpServletRequest Mock对象外，还需要模拟

ServletContext和

RequestDispatcher对象：

```
MockControl requestCtrl = MockControl.createControl(HttpServletRequest.class);  
HttpServletRequest requestObj = (HttpServletRequest)requestCtrl.getMock();  
MockControl contextCtrl = MockControl.createControl(ServletContext.class);  
final ServletContext contextObj = (ServletContext)contextCtrl.getMock();  
MockControl dispatcherCtrl =
```

```
MockControl.createControl(RequestDispatcher.class);  
RequestDispatcher dispatcherObj = (RequestDispatcher)dispatcherCtrl.getMock();
```

按照doPost()的语句顺序，我们设定Mock对象指定的行为：

```
requestObj.getParameter("username");  
requestCtrl.setReturnValue("admin", 1);  
requestObj.getParameter("password");  
requestCtrl.setReturnValue("123456", 1);  
contextObj.getNamedDispatcher("dispatcher");  
contextCtrl.setReturnValue(dispatcherObj, 1);  
dispatcherObj.forward(requestObj, null);  
dispatcherCtrl.setVoidCallable(1);  
requestCtrl.replay();  
contextCtrl.replay();  
dispatcherCtrl.replay();
```

然后，测试doPost()方法，这里，为了让getServletContext()方法返回我们创建的ServletContext Mock对象，我们定义一个匿名类并覆写getServletContext()方法：

```
LoginServlet servlet = new LoginServlet() {  
    public ServletContext getServletContext() {  
        return contextObj;  
    }  
};  
  
servlet.doPost(requestObj, null);
```

最后，检查所有Mock对象的状态：

```
requestCtrl.verify();  
contextCtrl.verify();  
dispatcherCtrl.verify();
```

运行JUnit，测试通过！

倘若LoginServlet的代码有误，例如，将context.getNamedDispatcher("dispatcher")误写为context.getNamedDispatcher("dispatcher2")，则测试失败，JUnit报告：

```
junit.framework.AssertionFailedError:  
  
Unexpected method call getNamedDispatcher("dispatcher2"):  
getNamedDispatcher("dispatcher2"): expected: 0, actual: 1  
getNamedDispatcher("dispatcher"): expected: 1, actual: 0  
  
at ...
```

完整的LoginServletTest代码如下：

```
/**  
 * LoginServletTest.java  
 * Author: Liao Xue Feng, www.javaee4dev.com  
 */  
package com.javaee4dev.test.mock;  
  
import javax.servlet.*;  
import javax.servlet.http.*;  
  
import org.easymock.*;  
  
import junit.framework.TestCase;
```

```
public class LoginServletTest extends TestCase {
    public void testLoginFailed() throws Exception {
        MockControl mc = MockControl.createControl(HttpServletRequest.class);
        HttpServletRequest request = (HttpServletRequest)mc.getMock();
        // set Mock Object behavior:
        request.getParameter("username");
        mc.setReturnValue("admin", 1);
        request.getParameter("password");
        mc.setReturnValue("1234", 1);
        // ok, all behaviors are set!
        mc.replay();
        // now start test:
        LoginServlet servlet = new LoginServlet();
        try {
            servlet.doPost(request, null);
            fail("Not caught exception!");
        }
        catch(RuntimeException re) {
            assertEquals("Login failed.", re.getMessage());
        }
        // verify:
        mc.verify();
    }

    public void testLoginOK() throws Exception {
        // create mock:
        MockControl requestCtrl = MockControl.createControl(HttpServletRequest.class);
        HttpServletRequest requestObj = (HttpServletRequest)requestCtrl.getMock();
        MockControl contextCtrl = MockControl.createControl(ServletContext.class);
```

```
final ServletContext contextObj = (ServletContext)contextCtrl.getMock();
MockControl dispatcherCtrl =

MockControl.createControl(RequestDispatcher.class);
RequestDispatcher dispatcherObj = (RequestDispatcher)dispatcherCtrl.getMock();
// set behavior:
requestObj.getParameter("username");
requestCtrl.setReturnValue("admin", 1);
requestObj.getParameter("password");
requestCtrl.setReturnValue("123456", 1);
contextObj.getNamedDispatcher("dispatcher");
contextCtrl.setReturnValue(dispatcherObj, 1);
dispatcherObj.forward(requestObj, null);
dispatcherCtrl.setVoidCallable(1);
// done!
requestCtrl.replay();
contextCtrl.replay();
dispatcherCtrl.replay();
// test:
LoginServlet servlet = new LoginServlet() {

public ServletContext getServletContext() {
return contextObj;

}
};
servlet.doPost(requestObj, null);
// verify:
requestCtrl.verify();
contextCtrl.verify();
dispatcherCtrl.verify();

}

}
```

总结：

虽然EasyMock可以用来模仿依赖对象，但是，它只能动态模仿接口，无法模仿具体类。这一限制正好要求我们遵循“针对接口编程”的原则：如果不针对接口，则测试难于进行。应当把单元测试看作是运行时代码的最好运用，如果代码在单元测试中难于应用，则它在真实环境中也将难于应用。总之，创建尽可能容易测试的代码就是创建高质量的代码。